Lektion 5

Diese Lektion zählt erfahrungsgemäß zu den schwierigeren im Lauf des Semesters. Bitte arbeiten Sie die Lektion konzentriert und in Ruhe durch und versuchen Sie, bis zur Besprechung in der Vorlesung gut vorbereitet zu sein, so dass verbleibende Fragen angesprochen und geklärt werden können.

Kernpunkt dieser Lektion ist es, wie der Begriff der Korrektheit genauer gefasst werden kann und wie formal begründet werden kann, dass ein Algorithmus korrekt funktioniert. Ein wichtiges Hilfsmittel wird dabei das Konzept der Schleifeninvariante sein. Es wird auch der Aspekt der Terminierung betrachtet, d.h. ob die Frage, ob das Programm auch nach endlicher Zeit beendet wird.

4.2.1 Korrektheitsaussagen

Soll eine Aussage über die Korrektheit eines Programmstücks gemacht werden, dann sind meist gewisse Einschränkungen zu machen, z.B. „Dieses Programmstück bewirkt, dass …, sofern am Anfang folgende Voraussetzung … erfüllt ist.“ Eine Korrektheits-aussage beseht somit immer aus einer sogenannten *Vorbedingung* für Voraussetzungen, die erfüllt sein müssen und einer *Nachbedingung,* die ausdrückt, was das Programmstück bewirkt.

Definition 4.7 - Korrektheitsaussage mit Vor- und Nachbedingung

- Eine (partielle) Korrektheitsaussage
  { *P* } *S* { *Q* }
  besteht aus einer Zusicherung *P*, der sog. *Vorbedingung* (engl. *precondition*), einem Programmstück *S* und einer Zusicherung *Q*, der sog. Nachbedingung (engl. postcondition).
- Ein partielle Korrektheitsaussage {*P*} *S* {*Q*} bedeutet: Wenn vor der Ausführung des Programmstücks *S* die Eigenschaft *P* gilt, dann ist garantiert, dass danach, sofern das Programmstück S zu Ende kommt, die Eigenschaft *Q* gilt.

Aufgabe 4.8 - Korrektheitsaussagen

- Welche der folgenden Korrektheitsaussagen treffen zu?
- `x`, `y`, `z` seien dabei ganzzahlige Programmvariablen, X_0, und Y_0 symbolische Konstanten.

(1)

```
{ y > 10 }
      if (x <= 0)
            y = y - x;
      else
            y = 2 * x;
{ y > 0 }
```

(2)

```
{ x < 10 }
      if (x < 0)
            y = y - x;
      else
            y = 2 * x;
{ y > 0 }
```

(3)

```
{ x = X0 ∧ y = Y0 ∧ y ≥ 0}
      z = x;
      while(y > 0) {
            z = z + 1;
            y = y - 1;
      }
{ z = X0 + Y0}
```

Anmerkungen

- Diese Form der Korrektheitsaussagen wird als *partielle Korrektheit* bezeichnet, da damit nicht ausgesagt wird, dass das Programm garantiert zu Ende kommen wird. Es gibt auch den Begriff der *totalen Korrektheitsaussagen*, die auch garantieren, dass die Berechnung enden wird, sofern die Vorbedingung erfüllt ist.
- Eine Korrektheitsaussage für ein Programmstück kann als eine Art Vertrag betrachtet werden: Wenn jemand das Programmstück nutzen möchte, um die Vertragsleistung zu erhalten, d.h. dass am Ende die Nachbedingung erfüllt ist, dann muss er als Gegenleistung dafür sorgen, dass am Anfang die Vorbedingung erfüllt ist. Im Bereich des Software-Engineerings gibt es eine Programmentwurfsmethode „Design by contract“ (https://de.wikipedia.org/wiki/Design_by_contract), die auf solchen Spezifikationen mit Vor- und Nachbedingungen basiert.

4.2.2 Korrektheit von Schleifen

In Aufgabe 4.8 - Korrektheitsaussagen waren die Teilaufgaben (1) und (2) relativ einfach zu beantworten, da es im Programm durch die Fallunterscheidung nur zwei mögliche Ausführungspfade gab, die dann jeweils geprüft werden mussten. Im dritten Beispiel geht das nicht so einfach, da das Programm eine Schleife enthält und die Schleife zu unendlich vielen Ausführungspfaden führen kann:

$\{ x = X_0 \wedge y = Y_0 \wedge y \geq 0\}$

```
z = x;
while (y > 0) {
      z = z + 1;
      y = y - 1;
}
```

$\{ z = X_0 + Y_0\}$

Die Schleife kann, je nach gegebenen Ausgangswerten, beliebig oft durchlaufen werden: nullmal, einmal, zweimal, dreimal, ... Für alle unendlich vielen Ausführungsmöglichkeiten muss gezeigt werden, dass die Aussage korrekt ist. Natürlich kann man nicht unendlich viele Möglichkeiten einzeln durchgehen und überprüfen. Wie kann man trotzdem die Korrektheit für alle Fälle zeigen?

Die Lösung dafür ist das Konzept der *Schleifeninvariante*, mit der man gewissermaßen alle möglichen Ausführungen der Schleife "auf einmal" behandeln kann. Man sucht sich dazu eine geeignete Eigenschaft, die unabhängig von der Anzahl der Schleifendurchläufe ist. *Invariant* heißt dabei, dass die Eigenschaft vor und nach jedem Schleifendurchlauf zutrifft. Dann ist es egal, wie oft die Schleife durchlaufen wird, nach Beendigung der Schleife wird diese Eigenschaft auf jeden Fall immer gelten.

Definition 4.9 - Schleifeninvarianten

Eine Zusicherung (Aussage über die Variablenwerte eines Programms) **Inv** ist eine **Schleifeninvariante** für eine gegebene Schleifen, wenn gilt:
(1) **Inv** ist wahr direkt vor der Schleife.
(2) Ist **Inv** wahr vor einem Schleifendurchlauf, dann gilt **Inv** auch nach dem Schleifendurchlauf.

Folge: Die Invariante gilt vor und nach jedem Schleifendurchlauf.
Somit gilt die Schleifeninvariante **Inv** auch dann noch, wenn die Ausführung der Schleife beendet wird, unabhängig von der Anzahl der Schleifenduchläufe

- Wir gehen hierbei davon aus, dass die Schleifenbedingung keine Seiteneffekte hat, d.h. nicht den globalen Zustand, z.B. die Werte von Variablen, ändert.

Für eine Schleifeninvariante Inv gilt also folgendes:

```
{Inv}                          (1)
while(B){
      {Inv ∧ B}                (2)
      …
      {Inv}                    (3)
}
{Inv ∧ ¬ B}                    (4)
```

(**1) Direkt vor der Schleife**: Die Invariante muss direkt vor der Schleife gelten.

(2) Beginn der Schleifenrumpfs: Da die Schleifeninvariant vor und nach jedem Schleifendurchlauf gilt, gilt sie auch, wenn der Schleifenrumpf begonnen wird. Wir wissen an der Stelle zusätzlich, dass die Schleifenbedingung B gilt - sonst würde der Schleifenrumpf nicht mehr ausgeführt werden.

(3) Ende des Schleifenrumpfs: Wenn Inv eine Invariante ist, dann muss sie auch wieder am Ende des Schleifenrumpfs gelten.

(4) Direkt nach der Schleife: Da die Invariante vor und nach jedem Schleifen-durchlauf gilt, gilt sie auch nach dem letzten Schleifendurchlauf. Wir wissen nach der Schleife auch, dass die Schleifenbedingung B nicht mehr gilt, da sonst die Schleife nicht beendet worden wäre.

Aufgabe 4.10 - Schleifeninvarianten

Das folgende Programm soll das Produkt $n \cdot k$ für ganzzahlige Werte n und k berechnen, wobei $n \geq 0$.

```
{ n ≥ 0 }
      prod = 0;
      x = 1;
      while (x ≤ n) {
          x = x + 1;
          prod = prod + k
      }
{ prod = n·k }
```

Welche der folgenden Zusicherungen sind Schleifeninvarianten für die while-Schleife?

(1) $x > 0$

(2) $x < n$

(3) $x \leq n+1$

(4) $prod = (x-1) \cdot k$

Nachweis der Korrektheit mittels Schleifeninvariante

Wie ist nun vorzugehen, wenn wir die Korrektheit eines Programmstücks mit Schleife zeigen wollen? Wir gehen davon aus, dass wir schon wissen, dass vor der Schleife die Eigenschaft (Vorbedingung) *P* gilt und sollen nachweisen, dass nach der Schleife die Eigenschaft (Nachbedingung) *Q* gilt.

{***P***}

```
while(B){
     … // Schleifenrumpf
}
```

{***Q***}

Weder die Eigenschaften *P* noch *Q* sind typischerweise Schleifeninvarianten. Die Herausforderung besteht darin, eine geeignete Eigenschaft **Inv** zu finden, die folgende drei Forderungen erfüllt:

(1) Aus der Vorbedingung *P* muss die Invariante **Inv** logisch folgen.

$$P \Rightarrow \textbf{Inv}$$

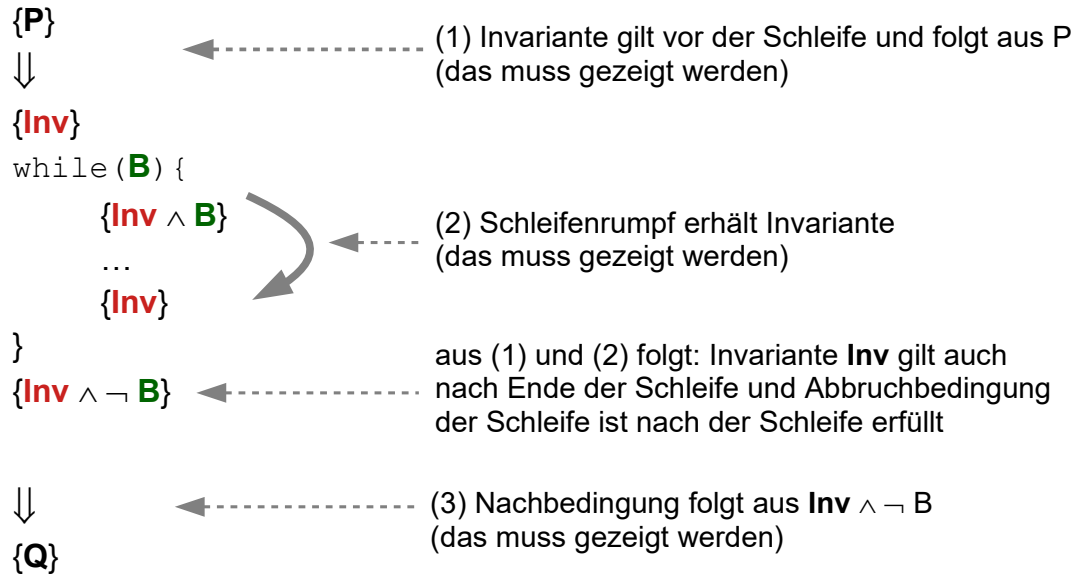
Das bedeutet, dass die Invariante gilt, wenn die Programmausführung die Schleife erreicht.

(2) Der Schleifenrumpf muss die Invariante erhalten, d.h. gilt die Invariante am Anfang des Schleifenrumpfs, dann muss sie auch wieder am Ende der Ausführung des Schleifenrumpfs gelten. Dazu kann auch verwendet werden, dass am Anfang des Schleifenrumpf die Schleifenbedingung *B* gilt, da der Rumpf nur ausgeführt wird, wenn die B erfüllt ist.

(3) Am Ende der Schleife muss gezeigt werden, dass aus der Invariante (die vor und nach jedem Schleifendurchlauf gilt und damit auch nach dem letzten Schleifendurchlauf) und der Abbruchbedingung $\neg B$ für die Schleife die zu beweisende Nachbedingung *Q* logisch folgt.

$$(\textbf{Inv} \wedge \neg B) \Rightarrow Q$$

Hier ist das nochmals zusammenfassend dargestellt:



Mit (1) und (2) weist man nach, dass **Inv** eine Invariante für die Schleife ist, also vor und nach jedem Schleifendurchlauf gilt.

Zu einer Schleife gibt es sehr viele unterschiedliche Schleifeninvarianten. Die Hauptschwierigkeit bei Korrektheitsbeweisen besteht darin, eine *geeignete* Schleifeninvariante zu finden, d.h. eine Invariante, so dass dann auch Schritt (3) möglich ist, man also zeigen kann, dass aus Schleifeninvariante und Abbruchbedingung die zu beweisende Nachbedingung *Q* folgt.

Wie findet man passende Schleifeninvarianten? Dazu gibt es keine Standardvorgehensweise. Hier ist Kreativität und Intuition gefragt und vor allem auch Verständnis dafür, wie das Programm funktioniert.

Beispiel 4.11 - Korrektheitsbeweis mit Schleifeninvariante

Es soll folgende Korrektheitsaussage bewiesen werden (x, y, z seien ganzzahlig).

$\{ x = X_0 \wedge y = Y_0 \wedge y \geq 0 \}$

```
z = x;
while(y > 0) {
      z = z + 1;
      y = y - 1;
}
```

$\{ z = X_0 + Y_0 \}$

Bevor wir uns an den Beweis machen, eine Anmerkung zu den Vor- und Nachbedingungen in diesem Beispiel: Um eine Zusammenhang zwischen Variablenwerten am Anfang und am Ende des Programms herstellen zu können,

werden hier in den Vorbedingungen symbolische Konstanten X_0 und Y_0 verwendet, die für unbekannte, aber feste Werte stehen. Die angegebene Korrektheitsspezifikation besagt also folgendes: Hat Variable x am Anfang einen Wert X_0 und Variable y am Anfang einen Wert Y_0, dann enthält Variable z am Ende X_0+Y_0, d.h. die Summe der Anfangswerte von x und y.

Doch nun zum Nachweis der Korrektheit. Was könnte eine geeignete Schleifen-invariante sein?

- Die Idee der Schleife ist grob gesagt die, dass man z jeweils um 1 hochzählt und y zugleich um 1 herunter, bis y bei 0 ist. Nimmt man die Summe y+z, erhält man also immer den gleichen Wert.
- Kann man noch genauer sagen, welchen Wert die Summe y+z hat? z hat am Schleifenbeginn den Wert von x, d.h. X_0, und y hat noch den Anfangswert Y_0.

1. Versuch:

Somit wäre ein erster Kandidat für eine Schleifeninvariante wie folgt:

Inv: $y+z = X_0+Y_0$

Damit versuchen wir jetzt die Korrektheit zu zeigen:

$\{ x = X_0 \wedge y = Y_0 \wedge y \geq 0\}$

```
z = x;
```

$\Downarrow$ (1)

```
{Inv:  y+z = X0+Y0 }
while(y > 0){
      {Inv:  y+z = X0+Y0 ∧ y > 0}
      z = z + 1;
      y = y - 1;
      {Inv:  y+z = X0+Y0}                (2)
}
{Inv:  y+z = X0+Y0 ∧ ¬ y > 0}
```

$\Downarrow$ (3)

$\{ z = X_0 + Y_0\}$

(1) Gilt die Invariante vor der Schleife? Ja, $y+z = X_0+Y_0$ gilt vor der Schleife, da z den Anfangswert X_0 von x zugewiesen bekommen hat und y noch den Anfangswert Y_0 hat.

(2) Gilt Invariante wieder am Ende des Schleifenrumpf, sofern die Invariante am Anfang des Rumpfs gegolten hat? Ja. $y+z = X_0+Y_0$ gilt wieder am Ende des Schleifenrumpfs, da am Anfang des Rumpfs $y+z = X_0+Y_0$ galt, Variable z um 1 erhöht und y um 1 vermindert wurde und somit die Summe gleich bleibt.

(3) Folgt aus der Invariante und Schleifenabbruchbedingung die Eigenschaft $z = X_0 + Y_0$?

$$y+z = X_0+Y_0 \wedge \neg y > 0 \quad \Rightarrow \quad z = X_0 + Y_0 \quad ???$$

Die Antwort ist leider *nein*! Inv ist zwar eine Invariante, aber leider keine geeignete Invariante für die zu zeigende Nachbedingung.

Als jemand mit Programmiererfahrung sagen Sie wahrscheinlich an dieser Stelle: y hat doch am Ende den Wert 0, also gilt $y+z = z = X_0+Y_0$. In (3) sehen wir aber (wenn wir rein formal argumentieren) dass wir bisher nur wissen, dass $\neg y > 0$ nach der Schleife gilt.

Warum wissen wir, dass y am Ende 0 ist? y wird heruntergezählt und die Schleife bricht ab, sobald y den Wert 0 erreicht. Damit haben wir auch schon einen Ansatzpunkt für den nächsten Versuch: Wir nehmen in die Invariante mit auf, dass y niemals negativ wird.

2. Versuch:

Wir erweitern die Invariante so:

Inv': $y+z = X_0+Y_0 \wedge y \geq 0$

Damit jetzt also ein neuer Versuch:

$\{ x = X_0 \wedge y = Y_0 \wedge y \geq 0\}$

```
z = x;
```

$\Downarrow$ (1)

$\{Inv': \; y+z = X_0+Y_0 \wedge y \geq 0\}$

```
while(y > 0){
```

$\{Inv': \; y+z = X_0+Y_0 \wedge y \geq 0 \wedge y > 0\}$

```
    z = z + 1;
    y = y - 1;
```

$\{Inv': \; y+z = X_0+Y_0 \wedge y \geq 0\}$ (2)

```
}
```

$\{Inv': \; y+z = X_0+Y_0 \wedge y \geq 0 \wedge \neg y > 0\}$

$\Downarrow$ (3)

$\{ z = X_0 + Y_0\}$

Was den Teil $y+z = X_0+Y_0$ der Invariante betrifft, gilt alles weiterhin wie im 1. Beweisversuch. Wir müssen nur den Teil $y \geq 0$ der Invariante noch prüfen:

(1) Gilt Invariante vor der Schleife? Ja, $y+z = X_0+Y_0$ s.o. und da $y \geq 0$ als Vorbedingung gegeben ist, trifft $y+z = X_0+Y_0 \wedge y \geq 0$ am Schleifenanfang zu.

(2) Gilt Invariante wieder am Ende des Schleifenrumpf? Ja. $y+z = X_0+Y_0$ gilt s.o. Da der Schleifenrumpf nur ausgeführt wird, wenn $y > 0$, also $y \geq 1$ am Anfang des Rumpfs erfüllt ist, ist garantiert, dass am Ende $y \geq 0$ gilt, da y im Durchlauf um 1 vermindert wird. Am Ende des Schleifenrumpfs gilt also wieder die Invariante $y+z = X_0+Y_0 \wedge y \geq 0$.

(3) Folgt aus der erweiterten Invariante und der Schleifenabbruchbedingung die zu beweisende Eigenschaft $z = X_0 + Y_0$? In der Invariante ist $y \geq 0$ enthalten, die Abbruchbedingung ist $y \leq 0$, d.h. $y = 0$ muss gelten.

$$y+z = X_0+Y_0 \wedge y \geq 0 \wedge y \leq 0$$
$$\Rightarrow \quad y+z = X_0+Y_0 \wedge y = 0$$
$$\Rightarrow \quad z = X_0 + Y_0$$

Somit ist nachgewiesen, dass z am Ende die Summe der Anfangswerte von x und y hat. □

Anmerkungen

- Die eher herantastende Vorgehensweise bei der Suche nach geeigneten Invarianten im Beispiel oben - man fängt mit einem ersten Kandidaten für eine Invariante an und bessert dann in folgenden Versuchen nach, wenn man feststellt, dass die Invarianten nicht ganz ausreicht - ist typisch für den Weg zu solchen Korrektheitsbeweisen.
- Das Konzept der Schleifeninvarianten macht erfahrungsgemäß am Anfang oft Verständnisprobleme. Zum einen muss erst einmal die grundlegende Vorgehensweise begriffen werden, zum anderen ist es sehr mühsam und aufwendig, das alles mathematisch formal und exakt hinzuschreiben.

  In der Praxis als Softwareentwickler wird man üblicherweise solche Beweise nicht mathematisch exakt durchführen und aufschreiben. Das Konzept der Schleifeninvarianten kann aber trotzdem sehr wichtig und hilfreich sein, um sich zu überlegen, ob ein Algorithmus, den man entwickelt (oder den man verstehen will), auch in allen Fällen richtig funktioniert. Die Grundschritte

  1. Überlege, was eine passende Invariante sein könnte.
  2. Überlege, ob die Invariante vor der Schleife gilt.
  3. Überlege, ob die Invariante auch nach einem Schleifendurchlauf wieder gilt.
  4. Überlege, ob aus der Invariante dann die gewünschte Eigenschaft nach der Schleife folgt.

  werden vom geübten und erfahrenen Software-Entwickler dann nur informell im Kopf durchgeführt.

Aufgabe 4.12 - Noch ein Korrektheitsbeweis

Überlegen Sie sich eine geeignete Schleifeninvariante, mit der die Korrektheit für das Programm aus Aufgabe 4.10 bewiesen werden kann.

```
{ n ≥ 0 }                                   // Vorbedingung
      prod = 0;
      x = 1;
      while (x ≤ n) {
          x = x + 1;
          prod = prod + k
      }
{ prod = n·k }                              // Nachbedingung
```

Als weiteres Beispiel für einen Korrektheitsnachweis mit Hilfe einer Schleifeninvariante betrachten wir das Sortierverfahren Insertionsort, das bereits vorgestellt wurde.

zu Aufgabe 4.5 - Korrektheit von Insertionsort

Wir wollen hier nur die äußere Schleife von Insertionsort (Algorithmus 4.4) betrachten und nehmen dabei an, dass wir uns schon überlegt hätten, dass die innere Schleife richtig funktioniert, d.h. den aktuellen Wert arr[j] an passender Stelle einsortiert und den sortierten Bereich dadurch um einen Wert verlängert.

```
static void insertionSort(double[] arr) {
   int j = 1;

   while (j < arr.length) {
      // Element arr[j] an richtiger Stelle in den sortierten Bereich a[0] bis a[j-1] einfügen
      …
    }
}
```

Am Ende soll das komplette Array `arr[0..arr.length-1]` sortiert sein. Als Vorbedingung können wir voraussetzen, dass das Array mindestens einen Wert enthält, d.h. `arr.length > 0`.

1. Versuch:

Die zuvor schon beschriebene Grundidee von Insertionsort, dass am Anfang des Felds ein sortierter Bereich aufgebaut wird, ist auch die Basis für eine Invariante.

Schleifeninvariante $\mathbf{Inv_{Ins}}$:

Die Werte im Bereich arr[0.. j-1] sind aufsteigend sortiert

```
static void insertionSort(double[] arr) {
   { arr.length > 0 }                                   Vorbedingung
   int j = 1;

     ⇓                                                  (1)
   {Inv_Ins}
   while (j < arr.length) {
      {Inv_Ins ∧ j < arr.length}
      // Element arr[j] an richtiger Stelle in den
      //  sortierten Bereich a[0] bis a[j-1] einfügen
      …
      j++;
    {Inv_Ins}                                           (2)
   }
    {Inv_Ins ∧ j ≥ arr.length }
     ⇓                                                  (3)
  { arr[0..arr.length-1] aufsteigend sortiert }         Nachbedingung
}
```

Wir gehen wieder die übliche vor:

(1) Gilt die Invariante **Inv_{Ins}** vor der Schleife? Da am Anfang j = 1 gilt, besteht arr[0..j-1] nur aus dem Eintrag arr[0]. Ein Wert für sich allein ist richtig sortiert.

(2) Gilt die Invariante **Inv_{Ins}** wieder am Ende des Schleifenrumpfs, sofern sie am Anfang des Rumpfs gegolten hat? Ja - die innere Schleife fügt den Wert arr[j] in den Bereich arr[0..j-1] richtig ein und vergrößert damit den sortierten Bereich um einen Wert. Variable j wird um 1 erhöht. Somit ist dann auch wieder der Bereich arr[0..j-1] sortiert.

(3) Folgt aus Invariante **Inv_{Ins}** und Schleifenabbruchbedingung, dass das ganze Array sortiert ist?

arr[0..j-1] ist aufsteigend sortiert ∧ j ≥ arr.length

Nein, leider lässt sich das nicht folgern.

Ähnlich wie im vorigen Beispiel ist unsere Invariante zu schwach, um das gewünschte Ergebnis nachweisen zu können. Die Invariante muss noch ein wenig verstärkt werden, so dass wir am Ende folgern können, dass Variable j genau den Wert `arr.length` hat.

2. Versuch: Wir nehmen zur Invariante dazu, dass j niemals größer als arr.length wird.

Neue Schleifeninvariante:

Inv_{Ins} ∧ j ≤ arr.length

(wobei **Inv_{Ins}: Die Werte im Bereich arr[0.. j-1] sind aufsteigend sortiert**, s.o.)

```
static void insertionSort(double[] arr) {
   {arr.length ≥ 1}                                          Vorbedingung
   int j = 1;
     ⇓                                                       (1)
   {Inv_Ins ∧ j ≤ arr.length}
   while (j < arr.length) {
      {Inv_Ins ∧ j ≤ arr.length ∧ j < arr.length}
      // Element arr[j] an richtiger Stelle in den
      // sortierten Bereich a[0] bis a[j-1] einfügen
      …
      j++;
    { Inv_Ins ∧ j ≤ arr.length }                             (2)
   }
    {Inv_Ins ∧ j ≤ arr.length ∧ j ≥ arr.length }
     ⇓                                                       (3)
  {arr[0..arr.length-1] aufsteigend sortiert }               Nachbedingung
}
```

Was wir für den sortierten Bereich im 1. Versuch überlegt haben, gilt weiterhin.

(1) Gilt Invariante **Inv'$_{Ins}$** vor der Schleife? arr[0..j-1] ist sortiert, (s.o. 1. Versuch)
Da nach der Zuweisung j = 1 gilt und nach Vorbedingung arr.length ≥ 1 gegeben ist, ist die Invariant "arr[0..j-1] ist sortiert ∧ j ≤ arr.length" am Anfang erfüllt.

(2) Gilt Invariante **Inv'$_{Ins}$** wieder am Ende des Schleifenrumpfs, sofern sie am Anfang des Rumpfs gegolten hat? Ja, arr[0..j-1] ist am Ende des Rumpfs sortiert (s.o. 1. Versuch)
Da beim Eintritt in den Schleifenrumpf j < arr.length gilt (Schleifenbedingung) und j in der Schleife um 1 erhöht wird, gilt am Ende der Schleife j ≤ arr.length.
Am Ende der Schleife gilt also die Invariante
arr[0..j-1] ist sortiert ∧ j ≤ arr.length
wieder.

(3) Folgt aus Invariante **Inv'$_{Ins}$** und Schleifenabbruchbedingung j ≥ arr.length, dass das ganze Array sortiert ist?
arr[0..j-1] ist aufsteigend sortiert ∧ j ≤ arr.length ∧ j ≥ arr.length
⇒ arr[0..j-1] ist aufsteigend sortiert ∧ **j = arr.length**
⇒ arr[0.. arr.length-1] ist aufsteigen sortiert

Somit ist also gezeigt, dass am Ende das ganze Array aufsteigend sortiert ist. □

4.3 Terminierung

Eine weitere Fragestellung im Zusammenhang mit der Korrektheit von Programmen ist die Fragen nach der *Terminierung*, d.h. danach, ob das Programm auch zu einem Ende kommen wird und nicht in Endlosschleifen und Endlosrekursion hängen bleibt.

Terminierungsproblem: Endet die Programmausführung mit zulässigen Eingabedaten in jedem Fall nach endlich vielen Schritten?

Partielle und totale Korrektheit

Die bisherige Argumentation mit Schleifeninvarianten garantiert nur die sog. *partielle Korrektheit*. Wir beweisen damit nur, dass das Ergebnis richtig sein wird, *sofern* das Programm terminiert. Wir haben damit aber nicht die sog. *totale Korrektheit* nachgewiesen, zu der auch gehört, dass garantiert ist, dass das Programm nach endlicher Zeit zu einem Ende kommen wird.

Beispielsweise könnten wir für folgendes Additionsprogramm, (ganz ähnlich dem von Beispiel 4.11, nur Schleifenbedingung $y \neq 0$ statt $y > 0$) formal beweisen, dass am Ende *z* die Summe X_0+Y_0 der Anfangswerte ist.

$\{ x = X_0 \wedge y = Y_0 \}$

```
      z = x;
      {Inv: z+y = X0+Y0 }
      while(y ≠ 0) {
            z = z + 1;
            y = y - 1;
      }
      {Inv: z+y = X0+Y0 ∧ y = 0}
```

$\{ z = X_0 + Y_0 \}$

Offensichtlich ist es aber so, dass dieses Programm für negative Werte von y zu einer Endlosschleife führt, also nicht terminiert.

Will man die totale Korrektheit zeigen, muss die Terminierung zusätzlich zur partiellen Korrektheit nachgewiesen werden. Bei den meisten Schleifen ist es recht einfach zu sehen, dass sie nach endlich vielen Durchläufen enden werden, z.B. bei typischen Zählschleifen

```
for (int i = 0; i < n; i++) {
      …
}
```

Die Frage nach der Terminierung kann aber auch alles andere als trivial sein. Betrachten Sie dazu folgendes Beispiel.

Beispiel 4.13 - Hasse-Collatz-Folge

Die Hasse-Collatz-Folge ist eine Folge von natürlichen Zahlen, die nach einem ganz simplen Prinzip gebildet wird. Man fängt mit einer natürliche Zahl *n* an.

- Wenn die Zahl gerade ist, wird sie halbiert.
- Wenn die Zahl ungerade ist, wird 3*n*+1 als neuer Wert genommen.

Sobald man die 1 erreicht, endet die Folge.

Fängt man z.B. mit *n* = 7 an, so erhält man folgende Folge:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Die Berechnung der Folge kann sehr einfach mit einer Schleife programmiert werden:

```
while (n > 1) {
   System.out.println(n);

   if (n % 2 == 0)
      n = n / 2;         //n halbieren, wenn n gerade
   else
      n = 3 * n + 1;     //neuer Wert 3n+1, wenn n ungerade
}

System.out.println(n);
```

Die Fragestellung (ursprünglich vom Mathematiker Lothar Collatz 1937 formuliert, später von seinem Kollegen Helmut Hasse publik gemacht) ist die, ob für jede natürliche Zahl *n* die Schleife terminieren wird, d.h. nach endlich vielen Schritten die 1 erreicht wird. (Wir gehen dabei davon aus, dass wir nicht mit endlicher Zahlendarstellung wie in Programmiersprachen, sondern mit unbegrenzten natürlichen Zahlen arbeiten).

Sie können einige Tests durchführen und werden feststellen, dass das Programm immer bei 1 endet. Aber das garantiert nicht, dass das für *jede* natürliche Zahl gilt.

- Wie würden Sie begründen, dass das Programm immer terminiert, d.h. dass immer die 1 erreicht wird?

Das sollte doch für so ein kleines, einfaches Programm keine allzu große Schwierigkeit sein. Erstaunlicherweise ist dieses Hasse-Collatz-Problem aber immer noch eines der ungelösten mathematischen Probleme! Es ist noch niemandem ein korrekter Beweis dafür gelungen, dass die Folge immer nach endlich vielen Schritten endet.

Es ist im Allgemeinen also nicht immer einfach zu beweisen, dass eine Programm nach endlicher Zeit terminieren wird.

- Wir betrachten nun noch einen anderen Algorithmus - neben Insertionsort auch eines der bekannten einfachen Sortierverfahren - genannt *Bubblesort*. Die

Grundidee bei Bubblesort ist die, das man das zu sortierende Array wiederholt durchgeht und dabei benachbarte Elemente vertauscht, die in der falschen Reihenfolge stehen.

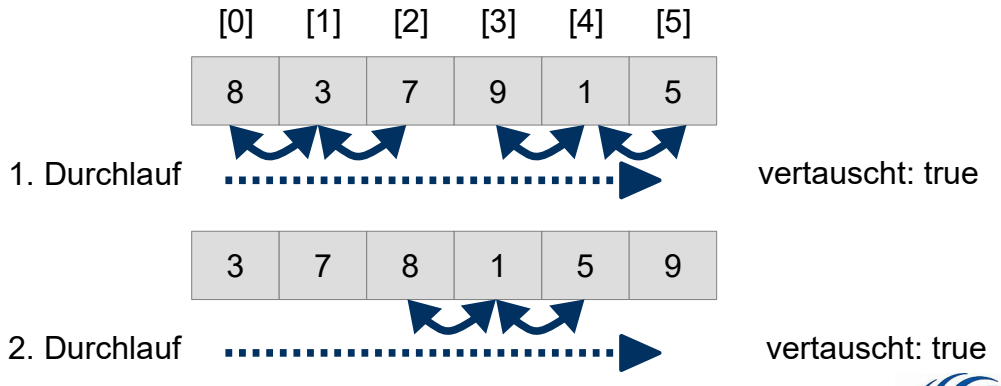
Algorithmus 4.14 - Bubblesort

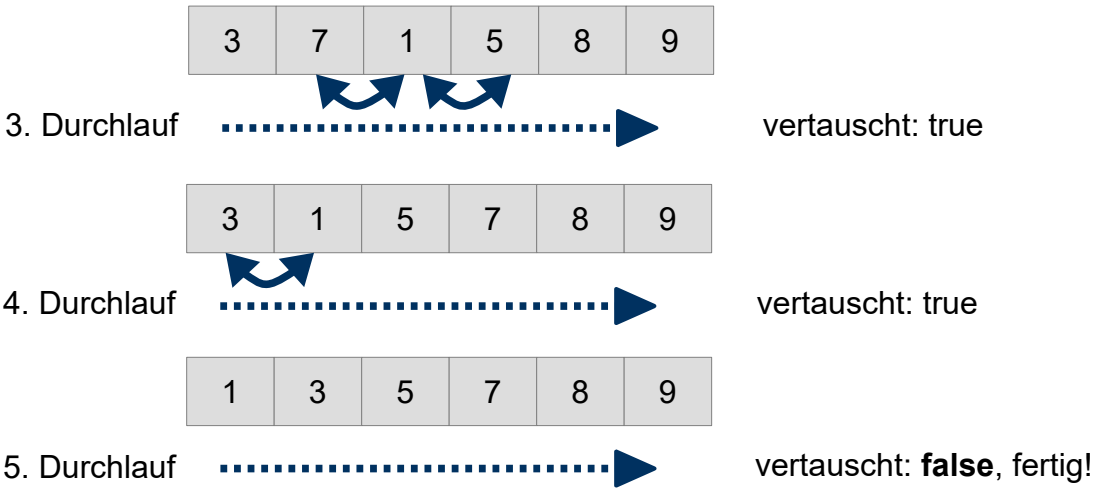
```
1 public static void bubbleSort(double[] a) {
2     boolean vertauscht;
3     do {
4         vertauscht = false;
5         for (int i = 0; i < a.length - 1; i++) {
6             if (a[i] > a[i+1]) {
7                 // a[i] und a[i+1] vertauschen
8                 double temp = a[i];
9                 a[i] = a[i+1];
10                a[i+1] = temp;
11                vertauscht = true;
12            }
13        }
14    }
15    while (vertauscht);
16}
```

- Das Array wird in der äußeren do-while-Schleife ggf. mehrfach durchlaufen. Bei jeder Wiederholung wird in der inneren for-Schleife (Zeile 5 – 12) das Array komplett durchlaufen und es werden jeweils zwei benachbarte Elemente miteinander verglichen. Sind sie in falscher Sortierreihenfolge, werden sie vertauscht. (Zeile 6 -12)
- Erfolgt bei einem Durchlauf keine Vertauschung mehr, ist das Sortierverfahren beendet (da dann alle Elemente in passender Reihenfolge sind).

Die Bezeichnung "Bubblesort" kommt daher, dass die größeren Werte durch die Vertauschungen nach und nach in Richtung zu den höheren Indices wandern, ähnlich wie Blasen im Wasser aufsteigen.

Beispiel 4.15 - Bubblesort





Korrektheit von Bubblesort

Bei Bubblesort ist die partielle Korrektheit recht einfach zu erkennen: Der Algorithmus terminiert erst dann, wenn einmal komplett das ganze Array durchgegangen wurde und alle Nachbarn in richtiger Ordnungsreihenfolge sind. Da immer nur Werte vertauscht wurden, ist dann das Array somit richtig sortiert.

4.3.1 Terminierungsnachweis für Schleifen

Wie kann man generell methodisch vorgehen, um zu zeigen, dass eine Schleife terminieren wird? Ein Ansatz dafür ist der folgende:

- Man sucht sich eine Eigenschaft (genannt *Terminierungsfunktion*), die, abhängig von den aktuellen Werten im Programm, einen ganzzahligen Zahlenwert liefert.
- Diese Eigenschaft muss so gewählt sein, dass der Wert der Terminierungs-funktion bei jedem Schleifendurchlauf um mindestens 1 kleiner wird.
- Man weist nach, dass die Schleife spätestens dann endet, wenn diese Eigenschaft eine bestimmte Untergrenzen (z.B. 0) erreicht.

Egal, welchen Wert *k* diese Eigenschaft am Anfang ergibt, nach endlich vielen Durchläufen wird man die Untergrenze 0 erreichen. Also kann die Schleife nicht endlos laufen.

Terminierungsnachweis mittels Terminierungsfunktion

Suche eine **Terminierungsfunktion $t(x_1,...,x_n)$**, wobei x_1 bis x_n die in der Schleife verwendeten Variablen sind.

- $t(x_1,...,x_n)$ muss immer einen **ganzzahligen Wert ≥ 0** als Ergebnis liefern, solange die Schleifeninvariante gilt.

- Bei jedem **Schleifendurchlauf** muss der Wert von **$t(x_1,...,x_n)$** um mindestens 1 **vermindert** werden (vorausgesetzt die Schleifeninvariante gilt)

Wird eine solche Terminierungsfunktion gefunden, dann ist garantiert, dass die Schleife nach endlich vielen Durchläufen terminiert.

Begründung

- am Anfang liefert $t(x_1,...,x_n)$ einen endlichen Wert *k* und die Schleifeninvariante gilt
- solange Schleife läuft, bleibt die Invariante erhalten und folglich ist $t(x_1,...,x_n) \geq 0$
- nach spätestens *k* Durchläufen muss die Schleife enden, da der Wert bei jedem Durchlauf um mindestens 1 vermindert wird und nie unter 0 sinken kann, solange die Schleife weiter läuft.

Beispiel 4.16- Terminierungsfunktionen

Wir betrachten folgenden Algorithmus, der den größten gemeinsamen Teiler von zwei positiven Zahlen nach dem Verfahren von Euklid berechnet.

```
// Vorbedingung: n > 0 und m > 0
int ggTEuklid(int n, int m) {
   while (n != m) {
      if (n > m)
         n = n - m;
      else
         m = m - n;
   }
   return n;
}
```

Ein Invariante, die sich für die Schleife nachweisen lässt, ist die, dass immer `n` > 0 und `m` > 0 gilt. Das wird sich nachfolgend als nützlich herausstellen.

Was wäre hier eine geeignete Terminierungsfunktion t(*n*, *m*), d.h. einen Eigenschaft, die von den Werten von `n` und `m` abhängt, bei jedem Durchlauf um mindestens 1 kleiner wird und immer größer-gleich 0 ist, solange die Schleife läuft?

Eine mathematisch einfach anzugebende Möglichkeit ist

$$t(n, m) = n + m$$

Sind die Eigenschaften einer Terminierungsfunktion erfüllt?

- t(*n*,*m*) liefert einen ganzzahligen Wert. Da die Schleifeninvariante $n > 0$ und $m > 0$ gilt, ist $t(n,m) = n+m \geq 0$, solange die Schleife läuft.

- Da $n > 0$ und $m > 0$ gilt, ist sichergestellt, dass bei jedem Schleifendurchlauf der Wert von t(n,m) um mindestens 1 kleiner wird, da entweder von n oder von m ein positiver Wert abgezogen wird.

Die erforderlichen Eigenschaften sind also erfüllt und somit ist nachgewiesen, dass die Schleife terminieren wird.

(Randbemerkung: Die Invariante, dass n und m positiv ist, ist dabei essentiell. Würde n oder m möglicherweise den Wert 0 erreichen, hätte man offensichtlich eine Endlosschleife.)

Aufgabe 4.17 - Terminierungsfunktion für Insertionsort

- Suchen Sie eine geeignete Terminierungsfunktion für die äußere Schleife von Insertionsort.

Zum Abschluss bleibt nun noch, die Frage nach der Terminierung von Bubblesort zu klären.

Aufgabe 4.18 - Terminierung von Bubblesort

Was aber bei Bubblesort nicht so einfach zu sehen ist, ist die Terminierung, d.h. ob der Algorithmus immer nach endlicher Zeit, d.h. nach endlich vielen Durchgängen der äußeren do-while-Schleife, zu einem Ende kommen wird - bei jeder Array-Länge und für jeden möglichen Array-Inhalt. Wie würden Sie das begründen?

Anmerkung - Terminierungsnachweise bei Rekursion

In ähnlicher Weise kann auch bei rekursiven Algorithmen nachgewiesen werden, dass sie terminieren.

- Dazu muss man sich eine Eigenschaft suchen, die von den Parameterwerten abhängt und bei jedem rekursiven Aufruf um mindestens 1 kleiner wird.
- Ist sichergestellt, dass die Rekursion spätestens dann beendet wird, wenn eine Untergrenze erreicht wird, kann es zu keiner Endlosrekursion kommen.

Fazit zu Lektion 5

Das sollten Sie nach dieser Lektion beantworten können

- Wie kann durch Vor- und Nachbedingungen das Verhalten eines Programms spezifiziert werden?
- Was ist eine Schleifeninvariante?
- Wie werden Schleifeninvarianten verwendet, um die Korrektheit von Schleifen zu zeigen?
- Was ist der Unterschied zwischen partieller und totaler Korrektheit?
- Wie kann durch eine Terminierungsfunktion gezeigt werden, dass ein Programm mit Schleife immer terminieren wird?